RC4-AccSuite: A Hardware Acceleration Suite for RC4-Like Stream Ciphers

RC4-AccSuite: A Hardware Acceleration Suite for RC4-like Stream Ciphers

Ayesha Khalid, Goutam Paul, *Senior Member, IEEE*, and Anupam Chattopadhyay, *Senior Member, IEEE*

***Abstract*—We present RC4-AccSuite, a hardware accelerator, which combines the flexibility of an ASIP and the performance of an ASIC for the most widely-deployed commercial stream cipher RC4 and its eight other prominent variants including Spritz (CRYPTO-2014 Rump-session). Our carefully-designed instruction set architecture reuses combinational and sequential logic at its various pipeline stages and memories, saving up to 41% in terms of area, compared to individual cores, a substantial throughput improvement and the power budget dictated primarily by the variant used. Moreover, using state-replication, noticeable performance enhancement of RC4 variants is achieved. RC4-AccSuite possesses extensibility for future variants of RC4 with little or no tweaking.**

***Index Terms*—ASIP, Hardware Accelerator, High throughput, RC4, Stream Cipher**

I. Introduction and Motivation

With the emergence of the pervasive computing paradigm, ensuring security for all the increased information exchange is becoming more and more challenging. Modern applied cryptography in communication networks requires secure kernels that also manifest into low cost and high performance realizations. The need of better performance justifies the efforts in the direction of design of high performance embedded Application Specific ICs (ASICs) dedicated to a certain cipher. Another critically required feature of these circuits, however orthogonal to the performance offered by the dedicated ASICs, is the need of making flexible designs. The need of of flexibility stems from the dynamic nature of cryptography, i.e., newer versions of algorithm suitable to newer platforms and or counteracting successful cryptanalytic attacks are frequently proposed. Flexibility could also be exploited to enable a user specified trade-off between security against system performance.

Since its inception 20 years back, RC4 has been the target of keen cryptanalytic efforts, some of which have been successful. Note that for cryptographic algorithms, there are two kinds of attacks. The first kind exploits the specific use of an algorithm in a protocol and therefore needs some assumptions to mount the attack. The second kind focuses on mathematical analysis of the algorithm without any application-specific assumptions. Most of the attacks on the RC4 stream cipher belong to the first kind. For example, in Wired Equivalent Privacy (WEP) protocol, the first 3 bytes of the secret key is used as public initialization vector (IV) and hence are known to the attacker. Thus, the WEP attacks [2] make use of this assumption. However, the actual specification of RC4 algorithm mandates no such requirements and the entire secret key remains private. Thus, the WEP attack strategy is not applicable to the base RC4 algorithm. Similar is the case with the attacks on other protocols like Wi-Fi Protected Access (WPA) and Transport Layer Security (TLS) [3] using RC4. None of these are applicable to the actual RC4 algorithm. Among the second kind of attacks, the best known key recovery attack recovers a 16-byte key from the knowledge of the secret internal state with a complexity more than 2^{53} [4]. But there is no direct key recovery attack on the exact RC4 algorithm from the knowledge of the keystream (which is actually observable under known plaintext attack model). The best known attack for RC4 state recovery from keystream has a complexity of 2^{241} [5]. Though Internet Engineering Task Force (IETF) is currently seeking replacement of RC4 in TLS protocol [6], it is interesting to note that the base RC4 algorithm is still cryptographically secure and can be safely used with proper precautions. In addition, there are more secure variants of RC4 in the literature such as RC4$^+$ [7], Spritz [8] etc. The usability of the *RC4-like* cipher kernels is re-iterated in the recent proposal of Spritz [8] from the authors of the original RC4. Apart from being a *drop-in* replacement for RC4, Spritz also offers an entire suite of cryptographic functionalities based on *sponge-like* constructive functions. As NIST SHA-3 competition declared a sponge-based kernel called *Keccak* [9] as the winner after a 5 year long competition, the usability, security and efficiency of sponge functions has been already been scrutinized and appreciated by the cryptanalytic community. Thus, even if RC4 is replaced by other stream ciphers in practical protocols, RC4 and its variants, with their elegant and robust structures, are likely to remain model stream ciphers for both designers and cryptanalysts for years to come.

The fact that RC4 has an entire class of well-known variants for ensuring higher security, better performance and versions for implementation on word-oriented platforms makes the study and design of a generic core for implementing RC4 and its variants worthwhile. RC4-AccSuite is an Application Specific Instruction set Processor (ASIP) whose instruction set architecture (ISA) is designed by identifying the common operation kernels of members of RC4-like stream ciphers family. The accelerator can switch to various RC4 variants

This is a substantially revised and extended version of the conference paper [1] by the second and the third authors. The detailed difference between the earlier paper and the current draft is described in the Appendix.

A. Khalid, The Centre for Secure Information Technologies (CSIT), Queen's University Belfast, UK, e-mail: a.khalid@qub.ac.uk;

G. Paul (*Corresponding Author*), Cryptology and Security Research Unit (CSRU), R. C. Bose Centre for Cryptology and Security, Indian Statistical Institute, Kolkata , India, e-mail: goutam.paul@isical.ac.in;

A. Chattopadhyay, School of Computer Engineering, Nanyang Technological University (NTU), Singapore, e-mail: anupam@ntu.edu.sg.

Manuscript received , ; revised , .

at run-time and gives the user the choice to choose a variant that matches his/her performance, security, power and platform need. This flexibility generally comes at the cost of lower throughput performance, however the design compensates for performance using the technique of memory replication. The resultant RC4-AccSuite boosts the flexibility of an ASIP and the performance of an ASIC.

A. RC4 Stream Cipher

RC4 was designed by Ron Rivest of M.I.T. for RSA Data Security in 1987. It is also known as ARC4 or Alleged RC4 since it remained a trade secret till its code was leaked in 1994 on Internet. The simplicity of its design and implementation attracted a lot of attention, making it one of the most widely deployed stream ciphers in industrial applications. Originally, it was considered primarily as a software stream cipher, however, due to the diversity and applicability of today's computing platforms, the boundary between software and hardware ciphers is fast fading away. Common use of RC4 is to protect Internet traffic using the Secure Sockets Layer (SSL), TLS, WEP, WPA etc. protocols along with several application layer softwares.

The RC4 algorithm was described in [10]. It has an internal state comprising of 256-byte array, denoted by $S[0 \dots N-1]$ and accessed by indices i and j. The three phases of RC4 operation (and most other stream ciphers) are the State Initialization (SI), Key Scheduling Algorithm (KSA) and the Pseudo Random Generation Algorithm ($PRGA$). Some stream ciphers require Initialization Vector Scheduling Algorithm ($IVSA$) phase after KSA as well. The secret key $k[0 \dots l-1]$ is expanded by repetition to a size equal to that of array S: $K[y] = k[y \bmod l]$, for $0 \leq y \leq N-1$. In every iteration of KSA and PRGA, i is incremented, j is updated, values of $S[i]$ and $S[j]$ are swapped while PRGA produces one byte of output which is XOR-ed with the one byte of the message to produce one byte of ciphertext (or plaintext in case of decryption). Other than these phases of operation, i.e., SI, KSA and PRGA, some variants of RC4 undergo another round of shuffling based on initialization vector or IV. This phase happens after KSA and is known as $IVSA$.

TABLE I
DESCRIPTION OF RC4

State Initialization (SI)
For i from 0 to $N-1$ in steps of 1 $S[i] \leftarrow i$; $j \leftarrow 0$;
Algorithm KSA
State Scrambling: For i from 0 to $N-1$ in steps of 1 $j \leftarrow (j + S[i] + K[i])$; Swap($S[i], S[j]$);

Algorithm PRGA
Indices Initialization: $i \leftarrow 0, j \leftarrow 0$; *Repeat for next plaintext byte*: $i \leftarrow i + 1$; $j \leftarrow j + S[i]$; Swap($S[i]$, $S[j]$); $t \leftarrow S[i] + S[j]$; Output $z \leftarrow S[t]$;

B. Variants of RC4

A compact description of some of the noticeable variants of RC4 to counteract cryptanalytic attacks follows (the list is not chronologically arranged but in decreasing order of similarity with RC4). The reader is kindly advised to refer to the respective references for a detailed description.

1) **RC4$^+$**: RC4$^+$ recommended complementary layers of computation during for KSA and PRGA phase on top of the original proposal of RC4 for achieving a better security margin [7]. These layers of computation achieve better scrambling and avoid key recovery attack during RC4$^+$ KSA and RC4$^+$ PRGA, respectively. Some intermediate VLSI design versions trading-off security against performance namely PRGA$^\alpha$ and PRGA$^\beta$ have also been undertaken [1].
2) **VMPC**: VMPC variant of RC4 is named so after a hard to invert VMPC function, used during KSA, IVSA and PRGA of VMPC variant of RC4 [11]. The VMPC function for an N variable permutation array named P, transformed into Q, requires a single modulo addition and three accesses of permutation state array as shown:
$$Q[x] = P[P[P[x]] + 1], where\ 0 \leq x \leq N-1$$
3) **RC4A**: RC4A was introduced to remove a statistical bias in consecutive bytes of PRGA in RC4 [12]. It uses two keys to carryout KSA into two arrays $S1$ and $S2$. Similarly, two indices $j1$ and $j2$ are used for $S1$ and $S2$ respectively during PRGA based on exchange shuffle model, inline with RC4 PRGA. The difference with the original RC4 is that here the index $S1[i]+S1[j]$ produces output from $S2$ and vice versa.
4) **RC4B**: A recent work exposed the vulnerability of both RC4 and RC4A to new new classes of statistical biases [13]. To overcome that, a new RC4 variant known as RC4B is introduced, which differs from RC4A only as it mixes the contents of the $S1$ and $S2$ during update of $j1$ and $j2$.
5) **RC4b**: A byte-variant of RC4 called RC4b was described in [14]. The author claimed to remove the known biases in RC4 by scuffling state elements twice and by explicitly discarding the first N bytes during KSA.
6) **NGG(n,m)**: NGG(n,m) is a word variant of RC4, extensible to 32/64 bit words with S much smaller than 2^{32}/2^{64} [15], where $S = 2^n$ is the size in words and m is the word size in bits ($n \leq m$). The SI for NGG uses a precomputed random array, the KSA and PRGA phases are similar to that of RC4, extended to words. NGG is named so after initials of its authors.
7) **GGHN**: GGHN is an improved version of NGG, also named so after its designers initials [16]. It recommends multiple iterations of KSA phase, depending on word size and number of word of S for maintaining a high degree of randomness. For better security a key dependent third variable k is also used, other than i and j for exchange shuffle model in GGHN PRGA.
8) **Spritz**: Spritz is the recent proposal, coming from the author of RC4, formulated as a sponge and consequently capable of being used as a block cipher, stream cipher, hash functions, DRBG, MAC and AE [8]. It has RC4-like general design principles and attempts to repair weak design decisions of RC4.

C. Previous Work and Motivation

In the context of flexible cryptographic implementations, the idea of resource sharing for exclusive execution of more than one modes or versions of cipher algorithms is not novel. The motivation of designing these flexible hardware co-processors (or weakly programmable ASICs) stems from the need of various cryptographic functions required for ensuring privacy, authenticity and integrity. For block ciphers, after the widespread acceptance and use of AES, many unified configurable cores for AES with other ciphers were proposed, e.g., AES-128 with block cipher ARIA [17], AES-128/192/256 and AES-extended [18], AES-128 and Camellia [19]. Similarly, stream ciphers ZUC and SNOW 3G were combined in an area-frugal single ASIC, since both of them were included in the LTE-advanced security portfolio [20]. This case study was extended for a unified implementation of stream ciphers HC-128 and RC4 in [21]. Efforts for the design for unified co-processors were extended to include hash functions along with the block ciphers, hereby providing confidentiality and authenticity, simultaneously, examples include AES and Grøstl [22], AES and Fugue [23].

More generic cryptographic processors include CryptoManiac [24], a flexible 4-core VLIW processor with a 32-bit instruction set. Based on an analysis of the considered cryptographic applications, the added instructions combined logical operations with arithmetic and memory operations taking one to three cycles. It provided moderate performance enhancement over a wide variety of algorithms. Cryptonite [25] was a crypto processor with a small specialized instruction set and a two cluster architecture. Since it was not based on an existing instruction set and was designed from scratch, it was light-weight in comparison to CryptoManiac [24] and had lesser register port pressure and reduced routing constraints. It combined up to three standard logic, arithmetic, and memory operations and outperformed CryptoManiac for many block ciphers and hash functions. Following the same lines, another proposal was CCproc [26], a simple 32-bit co-processor with an extended RISC instruction set and datapath structure. It had a 5-stage pipelined datapath and a specifically designed instruction set to improve processing of symmetric-key algorithm. It offered limited compound instructions but promised support for future cryptographic proposal due to its generality. These unified cores successfully achieve area efficiency, compared to the sum of individual cores, due to resource sharing. Moreover, the throughput penalty in most cases is small, when compared to the slower of the implemented algorithms, due to existence of a common critical path. A more recent configurable co-processor, CoARX, exploits operational similarity between cryptographic functions to implement different block ciphers, hash functions, stream ciphers that are based on ARX family of ciphers [27]. FPGA implementation of AES and Keccak multi functional cores have been compared with each other in [28].

D. Original Contribution

Our RC4-AccSuite outperforms in three respects as discussed below

1) **Flexibility**: The flexibility of RC4-AccSuite is demonstrated by the fact that other than the basic proposal of RC4, it can execute 6 well-known byte-variants of RC4, namely, RC4$^+$ [7], VMPC [11], RC4A [12], RC4B [13], RC4b [14], Spritz [8], and 2 of its word-variants, namely, NGG [15], GGHN [16].
2) **Performance**: For promising higher performance we systematically undertake the state replication technique and integrate it in the design individually for all RC4 variants. Consequently, the performance degradation due to a flexible design is compensated.
3) **Resource Minimization**: Both performance and flexibility is achieved in processors at the cost of resources. For RC4-AccSuite we identify the reusable resources between RC4 and all the variants, including registers, pipeline registers, combinational macros and memory blocks whenever possible and consequently, the resource budget of RC4-AccSuite is much smaller compared to the accumulation of individual cores of RC4 variants implementation.

To the best of our knowledge this is the first endeavor to develop an ASIP for a well-known cryptographic cipher family with all these three features together. Design and development of RC4-AccSuite is an extension of the proposal put forward for a unified core for RC4 and RC4$^+$ [7] in a single core [1], however that lacked a conscious effort for resource reuse except for where an entire instruction could be reused. RC4-AccSuite can switch to any of these RC4 variants as per the user requirements and has extensibility to accommodate future RC4 variants. We performed an incremental design of RC4-AccSuite, adding one RC4 variant in each step. All the intermediate design points were implemented using HDL and synthesized with 65nm CMOS technology. The immense saving in terms of area and the power budget for each of these cores has been documented. The rest of the paper is organized as follows. Section II explains the high level architecture and interfaces of our RC4-AccSuite. We review the performance enhancement techniques applied so far to RC4 implementations and specify the memory replication technique with a case study for RC4$^+$ in Section III. The resource economization is explained in detail in Section IV with merger of RC4 and RC4$^+$ core as an example. Section V explains the area, power and throughput results of various versions of RC4-AccSuite along with a comparison with existing work. Section VI concludes this paper and provides future roadmap.

II. High-level Architecture of RC4-AccSuite

Fig. 1 presents high-level architectural diagram for the processor executing RC4 variants. The architecture is generalized for the VLSI implementation of any stream ciphers with large internal states as it is provided with an external memory bank, e.g., WAKE, Py, HC-128/256, CryptMT etc. The processor core performing one or more variants of RC4 is referred as RC4-AccSuite. A program memory keeps the instructions, while a program counter (PC) serves as the memory address. A more sophisticated address control for supporting loops and jumps is excluded as it is not required. The internal pipeline

TABLE II
BYTE-WIDE MEMORY REQUIREMENTS ($instances \times depth$) FOR RC4 VARIANTS

RC4 Variant	keystream word (bits)	S memories ($instances \times depth$)	K	IV	Total (bytes)
RC4 [10]	8	1×256	1×32	-	288
RC4$^+$ [7]	8	1×256	1×32	-	288
VMPC [11]	8	1×256	1×32	1×32	320
RC4A [12]	8	2×256	2×32	-	576
RC4B [13]	8	2×256	2×32	-	576
RC4b [14]	8	1×256	1×32	1×32	320
NGG [15]	32	4×256	1×32	-	1056
GGHN [16]	32	4×256	1×32	-	1056
Spritz [8]	8	1×256	1×32	-	288

architecture of the processor core, its input/output interface and the external memory bank changes depending on the RC4 variant/variants it supports. The IOs of the core are discussed below.

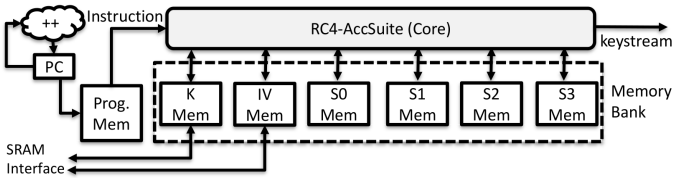

Fig. 1. Block diagram of RC4-AccSuite

1) *Instruction* is input to the core and is $\lceil \log_2(n) \rceil$ bits for n instructions specified for the RC4 variant. If the core supports two variants with $n1$ and $n2$ distinct instructions then the n is taken as $n = n1 + n2$.
2) *Keystream* is output of the core and its width is taken to be the maximum keystream word size of the supported variants. Hence for RC4/RC4$^+$ it will be 8 bits and for RC4/GGHN it will be 32 bits.

Due to large sizes of internal states (or S-Boxes) a preferred storage medium for the RC4 variants are the vendor supplied SRAMs which are optimized for throughput. The memory bank may include SRAMs for Key (K Memory, 32 words), IV (IV Memory, 32 words) and internal states ($S0 - S3$ Memories, 256 words), each being 8 bits-wide. An SRAM is selectively included in the memory bank provided at least one of the supported variants requires it, as given in Table II. K and IV need an external interface so that a new key and IV may be supplied from host processor before KSA and IVSA are initiated. RC4A keeps two internal S arrays and therefore requires both $S0$ and $S1$. For the two word variants, only NGG(8,32) and GGHN(8,32) are the currently supported configurations, hence S array has to have 256 words of 32 bits each. We instead use $S0 - S3$ in-order to reuse the same memory for byte-variants as well. For the rest of the discussion NGG(8,32) and GGHN(8,32) are referred as NGG and GGHN, respectively.

III. PERFORMANCE ENHANCEMENT FOR RC4-ACCSUITE

The performance of a crypto processor is usually benchmarked by the initialization time (KSA and IVSA if applicable) and the keystream generation throughput (byte/word/bits per second). We however use *cycles per keystream byte*, a more generic term. The rationale behind this originates from the fact that there will be different critical paths for mapping a processor design on different CMOS technology libraries. Thus, the maximum operating frequency and the throughput differs from one design to the next. Similarly, if the SRAM access time dictates the critical path of the processor, the performance will be dependant on the memory modules. Thus, considering cycles/keystream word/byte as the performance evaluation criteria seems more generic or technology independent. Nevertheless, in Section V, various design points of RC4-AccSuite are benchmarked on a CMOS technology library and the throughput results in typical parameters are discussed as well.

This section discusses the performance enhancement techniques undertaken in literature for RC4 and its variants implementation individually. We then extend the discussion for a more general implementation undertaken in RC4-AccSuite The previous work in this regard and throughput enhancement limit, wherever applicable, is also described.

A. Performance Enhancement for RC4 Variants

In one of the earliest RC4 implementations, Kitsos et al. [29] used 3 separate ported SRAMs for $S[i]$, $S[j]$ and $S[t]$ accesses. Due to data dependencies, 3 reads and 2 writes required per PRGA byte are distributed over three cycles, resulting in an encryption speed of 3 cycles/byte. A similar idea was published by Matthews [30]. We list the improvement techniques taken up for RC4 implementation; they are applicable to most of its variants as well.

1) **Using Multi-ported SRAMs**: The number of read/write ports of an SRAM restricts multiple simultaneous accesses and hence the algorithm throughput. The idea of using a multi-ported SRAM was first proposed by Matthews [30] who hinted a $5/3/1$ cycles per byte of RC4 PRGA throughput using a single, dual or 5 ported SRAM respectively, provided all the data currencies are being taken care of. Extending on these lines, throughput was improved to 2 cycles per byte using a tri-ported SRAM [31]. For RC4-AccSuite we choose dual ported SRAMs since they are the most common SRAM configuration having many vendors offering their optimized, low priced design.
2) **Loop Unrolling**: Unrolling of RC4 PRGA loop boosts throughput, but additional penalty is paid to counter possible Read After Write (*RAW*) hazards when the i^{th} and the $(i+1)^{th}$ loops write and read from the same S memory location, respectively. Sengupta et al. [32] unrolled the PRGA loop twice and reported a throughput up to 0.5 cycles per byte using a register-based storage in a pipelined RC4 circuit. For RC4-AccSuite, the loop unrolling was not employed since the additional hazards-avoiding checks arising due to simultaneous loop processing are algorithm-dependent and cannot be reused, resulting in an area-inefficient design.
3) **State Replication**: The use of multiple copies of S array with multiple SRAM instances enhances the availability of the simultaneous access ports and hence the

throughput, in line with Mathews proposal [30]. Using two dual-ported SRAMs for RC4 PRGA a 2 cycles per byte throughput was reported [1]. This idea was viably extended to other stream ciphers like HC-128 [33]. RC4-AccSuite supports word variants of RC4 requiring 4 internal byte-wide memories,consequently, state replication is justifiably applied to a byte-variants of RC4, whenever possible.

4) **State Splitting**: Splitting the large state array into smaller parts with known address distribution and keeping each smaller part in a separate dual ported memory enables more parallel accesses and in turn enables faster keystream generation. A $2\times$ and $4\times$ throughput enhancement by a 2-way and 4-way memory splitting is reported for HC-128 [34]. Unlike HC-128 that allows multiple parallel independent accesses in a PRGA, RC4 memory accesses are tied up to be sequentially performed due to possible Read After Write (*RAW*) hazards. To avoid these hazards, algorithm dependent extra checks for pipeline stalling would be required. To avoid the consequently data inefficient design, memory splitting was not exploited in RC4-AccSuite.

B. State Replication in RC4-AccSuite

State replication, using a dual ported SRAM enables 2 additional state accesses and consequently lowers clock cycles per PRGA, increasing throughput. For any RC4 byte-variant, memory replication to increase simultaneous state memory accesses is carried out provided no data incoherence arise as an aftermath. For an algorithm if a state memory S is replicated $m\times$ to achieve parallelism the implementation is dubbed as *ALGO_S_m*, e.g., RC4_S_0 and RC4_S_1 have been discussed in [1], [30] with a throughput of 3 and 2 cycles per byte, respectively.

A limitation in the context is noteworthy. Given *l*-SRAMs, each being *n*-ported, a critical question is that in *k*-cycles can all the $l \times n \times k$ access ports be utilized or not? The architecture of current SRAMs prohibits this maximum usage, as one or more clock cycles are required for *turnaround* to change the SRAM access between read and write. As a result, only two consecutive reads or two consecutive writes can be performed in two consecutive clock cycles from each access port of an SRAM. If one considers an SRAM requiring a single turnaround cycle, then for write immediately followed by read or vice versa from one access port, one clock cycle is required between the two accesses. Regarding efficient use of memory ports in PRGA or any RC4 variant following guidelines can help in achieving better performance.

1) Schedule writes on one port of the memory only and reads on both ports. Only one port will be underutilized by turnaround cycle and not both.
2) Schedule multiple writes on same memory in consecutive cycles if possible, hence turnaround cycle waste will be per batch of writes and not every write.

Memory replication may considerably boost throughput at the cost of additional area and power. To economize power, memory replication should be skipped, even if memory modules are available on a platform, Memory replication also costs additional writes to keep all state copies updated with correct data during KSA, IVSA and PRGA for RC4 and variants, e.g., RC4_S_0 and RC4_S_1 require 1 and 2 writes per PRGA. Memory replication should be incrementally applied on an algorithm followed by a systematic cycle by cycle design re-evaluation exploiting additional parallelism. Various interesting design points may arise, with performance-area-power trade-off. The next section gives a walk-through into two case studies for Spritz and RC4$^+$ for increasing throughput using state replication of SRAM .

TABLE III
PRGA FOR SPRITZ (TOP) AND RC4$^+$ (BOTTOM)

Spritz PRGA
Output Keystream Generation Loop:
$i = i + w$;
$j = k + S[j + S[i]]$;
$k = i + k + S[j]$;
Swap($S[i]$, $S[j]$);
Output $z = S[j + S[i + S[z + k]]]$;

RC4$^+$ PRGA
Output Keystream Generation Loop:
$i = i + 1$;
$j = j + S[i]$;
Swap($S[i]$, $S[j]$);
$t = S[i] + S[j]$;
$t' = (S[i_R^3 \oplus j_L^5] + S[i_L^5 \oplus j_R^3]) \oplus\ 0xAA$;
$t'' = j + S[j]$;
Output $z = (S[t] + S[t']) \oplus S[t'']$;

1) Spritz: Table III gives the PRGA steps for Spritz. It requires 6 state read accesses and two writes per PRGA byte generated. Fig 2 shows the mapping memory accesses when the Spritz PRGA steps are mapped on a dual ported SRAM (no replication). Due to data dependencies, no pipeline stage entertains more than one memory read, although 2 simultaneous request are possible. There should be 5 (or more) *nop* instructions between two consecutive instructions so that the cycle 1 of next instruction overlaps with the cycle 7 of current instruction resulting in a throughput of 6 cycles/byte. Further overlap of cycles for consecutive instructions is not possible due to the structural hazard caused due to availability of limited number of access ports of the memory.

Fig 3 shows memory accesses for Spritz_S0_1. This replication requires 2 additional memory writes as indicated in DP3 and DP4 pipeline stages. The last three reads for output calculation of Spritz PRGA are directed to $S1$. This enables the next consecutive instruction execution after 3 *nops*. The throughput is improved to 4 cycles/byte.

By carefully placing the accesses on the memory ports we ensure that overlap of consecutive instructions causes no data incoherence/ resource hazard. Fig 4 depicts one PRGA byte generated after every 4 cycles. Further parallelization through memory replication is not considered, since data dependencies in the algorithm do not allow it.

2) RC4$^+$: For RC4$^+$ PRGA, given in Table III (bottom), the relevant simplistic RC4$^+$_S_0 memory access mapping is given in Table IV. Out of the 4 extra reads here compared to PRGA RC4, two are are initiated in cycle 4 and two in

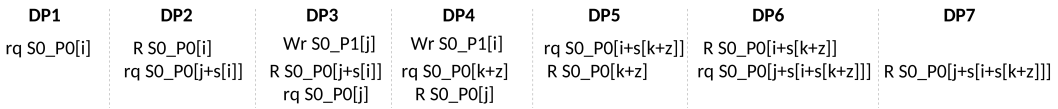

Fig. 2. Spritz PRGA mapped on one dual ported SRAM (P0= port 0, P1= port 1)

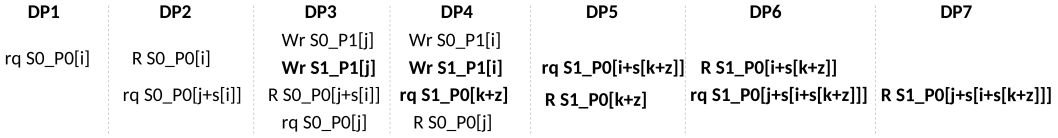

Fig. 3. Spritz_S0_1 mapped on two dual ported SRAM (P0= port 0, P1= port 1)

Fig. 4. Consecutive instructions overlap for Spritz_S0_1 without hazards

cycle 5. Due to bus turnaround it is not possible to initiate a read on port 1 in cycle 3, hence in cycle 4 only one access is possible. For simplicity we dub $S[i_R^3 \oplus j_L^5]$ and $S[i_L^5 \oplus j_R^3]$ as $S[t'^1]$ and $S[t'^2]$, respectively. There should be 5 (or more) *nop* instructions between two consecutive PRGA RC4$^+$ instructions so that no structural hazard is caused due to availability of limited number of access ports of the memory. A throughput of 5 cycles/byte results.

TABLE IV
RC4$^+$_S0_0, THROUGHPUT = 5 CYCLES/BYTE

cycles		1	2	3	4	5	6
S0	P0	rq S0[i]	R S0[i] rq S0[j]	R S0[j] rq S0[t]	R S0[t] rq S0[t'^{1}]	R S0[t'^{1}] rq S0[t"]	R S0[t"]
	P1		W S0[j]	W S0[i]	rq S0[t'^{2}]	R S0[t'^{2}] rq S0[t']	R S0[t']
R/W		0	2	2	1	2	2

We try out higher replication versions of RC4$^+$ for throughput enhancement. RC4$^+$_S0_1 and RC4$^+$_S0_2 improve the throughput to 4 and 3 cycles/byte, respectively. Table V (top) shows that for RC4$^+$_S0_2, only during 3rd cycle are all the 6 available simultaneous ports being used. There still is room for improvement that can be seen as an overlap of cycle 2 and cycle 4 of consecutive RC4$^+$ PRGA instructions is possible provided we had enough access ports. Using a total of 3 memories its is not possible due to number of operations being more than the access ports. RC4$^+$_S0_3 further improves the throughput to 2 cycles/byte as shown in Table V (bottom). By carefully placing the accesses on the ports we ensure that no more than one *nop* is required between two consecutive RC4$^+$ PRGA instructions. Overlapping of 2^{nd} and 4^{th} cycle also does not cause any data incoherence since a read priority is set for all SRAMs. Further parallelization through memory replication is not considered as RC4$^+$_S0_3 uses all the 4 S memories available, along with key and IV memories.

3) State Replication in RC4 Variants: For mapping any of the rest of the RC4 variants on the RC4-AccSuite the strategy followed is the same as described for RC4$^+$. First with replication factor of 0 we try to place all accesses in a chronological order unless disturbing it helps in utilizing an available port, carefully checking for data incoherence. Next we remap the accesses with a replication factor of 1, with doubled writes and keep doing so as long as a throughput boost is achievable or all the 4 memories are utilized. Table VI describes the effect on performance of these efforts. For VMPC, the parallelization cannot be exploited due to the dependency of reads on the previous value being read in the VMPC function. RC4A and RC4B require 2 S memories, replication of both, done once, occupies the 4 memories available. It generates two bytes simultaneously after each PRGA and parallelization by memory replication further improves the throughput to 1 byte

TABLE V
RC4$^+$_S0_2, THROUGHPUT = 3 CYCLES/BYTE (TOP), RC4$^+$_S0_3, THROUGHPUT = 2 CYCLES/BYTE (BOTTOM)

cycles		1	2	3	4
S0	P0	rq S0[i]	R S0[i] rq S0[j]	R S0[j] rq S0[t']	R S0[t']
	P1		W S0[j]	W S0[i]	
S1	P0		rq S1[t'^{1}]	R S1[t'^{1}] rq S1[t]	R S1[t]
	P1		W S1[j]	W S1[i]	
S2	P0		rq S2[t'^{2}]	R S2[t'^{2}] rq S2[t"]	R S2[t"]
	P1		W S2[j]	W S2[i]	
R/W		0	4	6	3

cycles		1	2	3	4
S0	P0	rq S0[i]	R S0[i] rq S0[j]	R S0[j]	
	P1		W S0[j]	W S0[i]	
S1	P0		rq S1[t'1]	R S1[t'1] rq S1[t]	R S1[t]
	P1		W S1[j]	W S1[i]	
S2	P0		rq S2[t'^{2}]	R S2[t'^{2}] rq S2[t"]	R S2[t"]
	P1		W S2[j]	W S2[i]	
S3	P0			rq S3[t']	R S3[t']
	P1		W S3[j]	W S3[i]	
R/W		0	5	7	3

per clock cycle. For the word variants, the parallelization is not possible since these algorithms use all the 4 S memories. For both the word variants the throughput is specified in terms of cycles per word (4 bytes) and the read write accesses are also for 32 bits (all four memories accesses simultaneously).

Out of the various memory replication versions of an algorithm as given in Table VI, we map the fastest one on RC4-AccSuite. For the rest of the discussion, we skip the postfix of an algorithm showing parallelization for simplicity (RC4$^+$_S0_3 is dubbed as RC4$^+$).

TABLE VI
ARRAY REPLICATION IN RC4 VARIANTS (C/B IS CYCLES/BYTE)

Algorithm	Replication factor	S Memories	Throughput C/B	Reads, Writes
RC4	-	1	3	3, 2
	S0_1	2	2	3, 4
RC4$^+$	-	1	5	7, 2
	S0_1	2	4	7, 4
	S0_2	3	3	7, 6
	S0_3	4	2	7, 8
VMPC	-	1	7	5, 2
RC4A/ RC4B	-	1	3/2	6, 4
	S0_1_S1_1	2	2/2	6, 8
RC4b	-	1	3/2	3, 2
	S0_1	2	2/2	3, 4
NGG	-	4	3/4	3, 3
GGHN	-	4	3/4	3, 1
Spritz	-	1	6	6, 2
	S0_1	2	4	6, 4

IV. RESOURCE ECONOMIZATION IN RC4-ACCSUITE

This sections talks about the potential resource sharing when two or more RC4 variants are clubbed together in RC4-AccSuite. We undertake the case study of an incremental build, starting with the design for RC4, then adding functionality for RC4$^+$ on top.

A. RC4-AccSuite Architecture (for RC4)

RC4-AccSuite has a *pipelined* architecture. It is equipped with a set of 8-bit ALUs with data registers, pipeline register and memory bank. The memory bank comprises of S0 and S1 for mapping RC4_S0_1, other than K and program memory. The processor has a 6 stage pipeline, out of which the last 4 are datapath for RC4 for completing its PRGA stage ($DP1$-$DP4$). For accommodating other variants with RC4 more stages for datapath are required.

The instruction set has 6 instructions for RC4, given in Table VII. The *nop* instruction serves to relieve structural hazards due to limited ports in the processor in between multiple *KSA* and *PRGA* instructions. The *set_regs0* and *set_regs1* instructions set the initial value of $counter$ register to be 0 and 1, respectively. The former is required before the start of SI and KSA phase while the later is required before PRGA. Fig. 5 shows the opcodes for these instructions as the selection of multiplexers (shown in bold font).

Fig. 5 represents the pipeline architecture of RC4-AccSuite. The memory accesses are shown as vertical arrows, read requests and writes with arrowheads pointing down while reads as pointing up. The I/Os of memory accesses are not shown to avoid unnecessary complexity. The operation division of the 6 pipeline stages is explained.

TABLE VII
INSTRUCTION SET FOR RC4

Instruction	Opcode	Comment
nop	0x0	No operation
set_regs0	0x1	Initializing registers, counter=0
set_regs1	0x2	Initializing registers, counter=1
init_S	0x3	S memory initialization (SI)
KSA	0x4	KSA phase
PRGA	0x5	PRGA Phase

1) *FE*: *Fetch* instruction stage uses an auto increment program counter (PC) register to access the synchronous program memory (PC). No jumps are supported.
2) *DI*: *Decode and Initialize* stage decodes the instruction and initializes $counter$ and j register as per the instruction. For *init_S* instruction, an increment of 2 is required for $counter$ register for writing to two memory locations using dual porting memories. For *KSA* and *PRGA*, increment by 1 is required.
3) *DP1*: First datapath stage for RC4. For *init_S* instruction, the two S memories are filled with the current value of $counter$ (assigned to a pipeline register i) and its one incremented value in two consecutive locations of memory using both ports. Hence for filling the 256 location only 128 instructions are required. For *KSA* and *PRGA* instructions, reads to $S0$ and K are also initialized.
4) *DP2*: For *KSA* the values read from the K and $S0$ memory are added to the j register and its value is updated. For *PRGA*, the j register update does not require value read from K memory. Both for *KSA* and *PRGA* instructions the value read at S[i] is written to both S memories with updated j as the address. For these instructions, a read is also requested for the same location on $S0$ memory. Due to a read before write priority no conflict results.
5) *DP3*: The second write for RC4 exchange shuffle model takes place for *KSA* and *PRGA* instructions. S[j] is written to index i using port 1 of both memories. The sum of S[j] and S[i] is calculated for *PRGA* instruction and a read to memory with t as index is initiated.
6) *DP4*: Only *PRGA* instruction requires this stage to read keystream word (S[t]).

B. Case Study: RC4$^+$ in RC4-AccSuite

To accommodate RC4$^+$ in addition to RC4 in RC4-AccSuite, additional resources are added only if the existing logic cannot be reused. In terms of additional memories, RC4$^+$_S0_3 requires $S2$ and $S3$ memories as well as an IV memory. The ISA requires 7 additional instructions (in addition to the RC4 instructions in Table VII), given in TableVIII. For initialization phase, only S0 and S1 are used. The replication is used to boost throughput only during *PRGA*$^+$, hence for better energy utilization the actual replication is delayed till

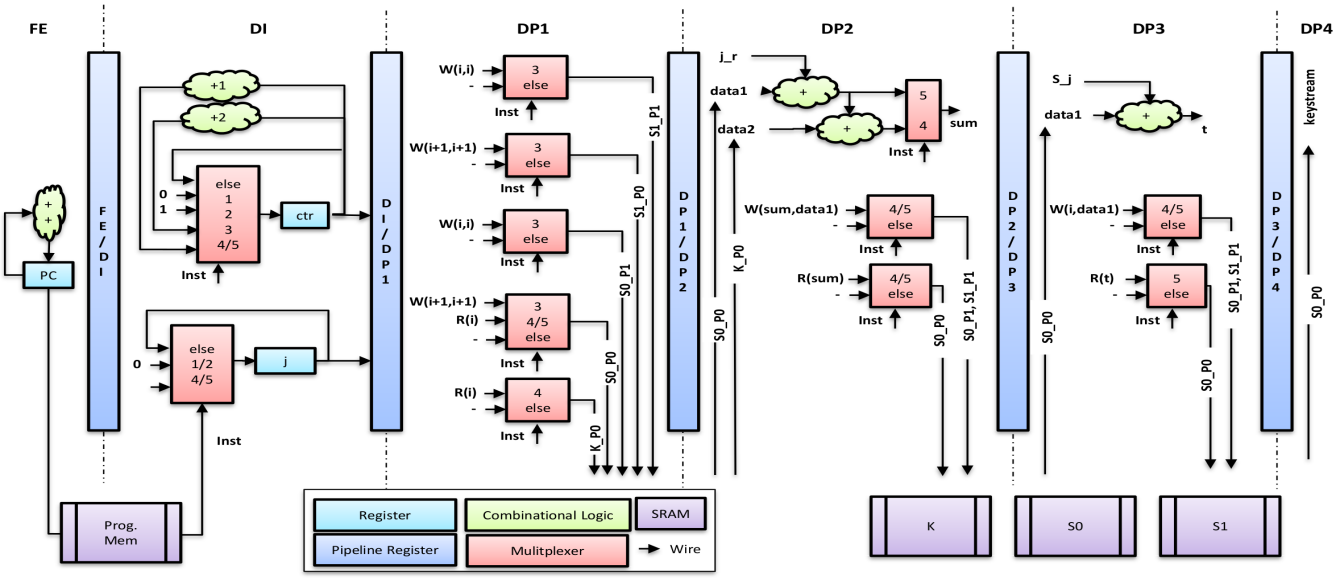

Fig. 5. Pipeline for RC4-AccSuite supporting only RC4

the last layer of RC4$^+$ KSA. The first layer of RC4$^+$ KSA is the same as RC4 KSA, hence no new instruction is added. The second layer of RC4$^+$ KSA is requires two iterations, *KSA_2a* and *KSA_2b* while the third layer implementation instruction is *KSA_3*.

TABLE VIII
INSTRUCTION SET EXTENDED FOR RC4$^+$

Instruction	Opcode	Comment
set_regs2	0x6	initializing registers, counter=127
set_regs3	0x7	initializing registers, counter=128
KSA_2a	0x8	KSA phase 2a
KSA_2b	0x9	KSA phase 2b
KSA_3	0xA	KSA phase 3
PRGA$^+$	0xB	PRGA Phase

Fig. 6 shows the pipeline for RC4-AccSuite capable of executing both RC4 and RC4$^+$. Here all resources shown in Fig. 5, that are being completely reused for RC4$^+$, have been shown in gray, while the additional resources to accommodate RC4$^+$ has been shown in color, following the same convention as given in the legend of Fig. 5. Most of the multiplexers size has been increased to accommodate the new set of instructions. The reused Instruction opcodes for RC4 has been shown in gray font while the new additions are shown in black. As can be seen the *FE* stage is completely reused. The *DI* stage is however reused partially since additional logic for various KSA instructions of RC4$^+$ is added. The additional instructions include *KSA_2a* and *KSA_2b* requiring *counter* initialization with 127 and 128 respectively, requiring two instructions *set_regs2* and *set_regs3* for register initialization. A decrementing counter required for *KSA_2a* and *KSA_3* is also supported.

The next four datapath pipeline stages for RC4$^+$ PRGA can be tallied with the memory accesses as given in Table V. All the resources of *DP1* stage are almost completely reused for RC4$^+$ except the additional read request from IV memory, that was not required previously for RC4. The second datapath stage, *DP2*, requires memory replication into S2 and S3 using port 1 for *KSA_3* and *PRGA*$^+$ as shown in Fig. 6. Moreover, for *KSA_2a* and *KSA_2b*, j register updating is carried out with one additional 8 bit adder and a XOR. For *PRGA*$^+$, t' calculation requires two intermediate reads, t'^{1} and t'^{2}, from register j and pipeline register i.

In *DP3*, calculation of t is reused as for RC4, however additional logic is required for calculating t' and t''. Respective multiple simultaneous reads are initiated using port 0 of S1, S2 and S3, thanks to the high replication factor. In *DP4* datapath stage S[t], S[t'] and S[t''] are read from S1_P0, S2_P0 and S3_P0, respectively and the keystream byte for RC4$^+$ is calculated after an eight bit addition and XOR-ing.

C. Instruction Datapath Reuse

Accommodating other RC4 variants with the existing pipeline structure of RC4-AccSuite required newer instructions and even additional pipeline stages. The VMPC PRGA instruction required additional pipeline stages to accommodate multiple interdependent memory accesses. Similarly, for Spritz, the 4 DP pipeline stage don't suffice, consequently, it has 7 DP pipeline stages as shown in Fig. 4 .

1) Entire Instruction Datapath Reuse: Most of the RC4 variants reuse instructions that are part of RC4 instruction set. Two such instructions are *nop* and *set_regs0* which are required by all the RC4 variants and hence their entire DP pipeline stages are reused. Both *set_regs0* and *set_regs1* instructions are entirely reused by all RC4 variants except VMPC.

2) Partial Instruction Datapath Reuse: Whenever possible, pipeline datapath reuse is maximized, within instructions of one algorithm or different algorithms and even if it is possible for few pipeline stages only. The *KSA* and *PRGA* instructions reuse the 8 bit adder in *DP4* as shown in Fig. 5. Similarly, *PRGA* and *PRGA*$^+$ instructions share the calculation and

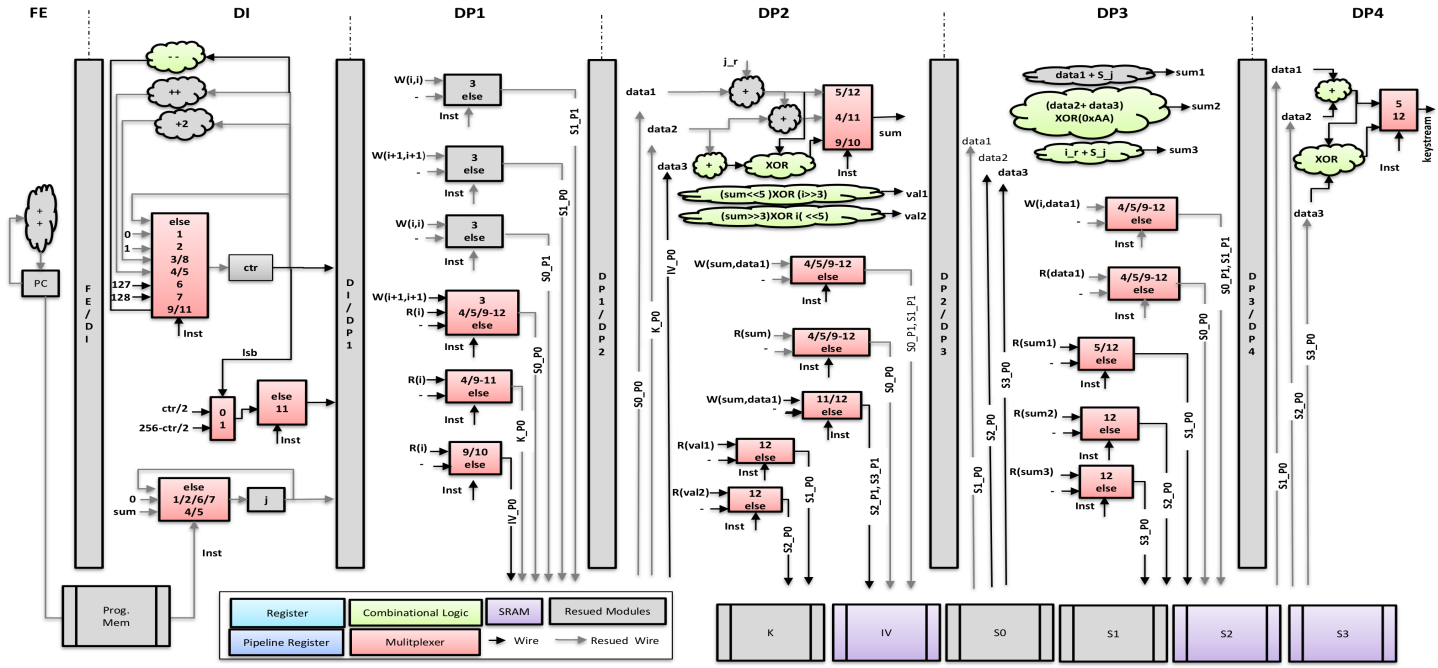

Fig. 6. Pipeline for RC4/RC4$^+$-AccSuite supporting both RC4 and RC4$^+$

reading of S[t] in the four DP pipeline stages, as seen in in Fig. 6 As RC4A is a parallelized version of RC4, the logic for RC4 KSA and PRGA in all the pipeline stages is reused (except for the j register update).

D. Registers/ Memories Reuse

For area economization, an aggressive reuse of registers, pipeline registers and memory modules is carefully designed and enabled when undertaking any RC4 variant. Since a pipelined register has more overhead than a register, its use should be carefully justified. A processor with $(n+1)$ pipelines may have each pipeline register replicated up to n times. Table IX shows the register/ pipeline registers and memory modules reuse in RC4-AccSuite. Please note that memory use is given as per the highest memory replication factor for each algorithm (refer Table VI).

- **Registers reuse**: RC4 requires only three registers for its execution: a program counter or *PC*, an incrementing/decrementing *counter* for keeping track of loop iterations, and index *j*. It is noteworthy that these registers are used by all the RC4 variants. Also, integrating RC4$^+$, VMPC and NGG into the RC4-AccSuite requires no additional registers. A second index variable *j2* is required by three RC4 variants: RC4A for enabling two parallel PRGAs, RC4b KSA requiring an index other than *j* (named as t in [14]) and Spritz (for holding keyword z). A 32 bits register *k* is required only for GGHN during KSA and PRGA for holding an intermediate value.
- **Pipeline Registers reuse**: RC4 requires three pipeline registers. First is an instruction register *IR*, optimized to the smallest width by Synopsys processor designer based on the total number of instructions, Secondly, index *i* that takes up the *counter* register value and is altered/ read by multiple pipeline stages during KSA and PRGA. These pipeline registers are reused by all RC4 variants as shown in Table IX. The third pipeline register *Sj* used by RC4 is required to hold S[j], read in *DP2* and added to S[i] in *DP3* during *PRGA*. In case of VMPC, the PRGA byte is generated before the swapping of S[i] and S[j].

TABLE IX
REGISTERS/ PIPELINE REGISTERS/ MEMORIES REUSE FOR RC4 VARIANTS

Resource	RC4	RC4$^+$	VMPC	RC4A/B	RC4b	NGG	GGHN	Spritz
Registers: Name (width)								
PC (8 bits)	✔	✔	✔	✔	✔	✔	✔	✔
counter (8 bits)	✔	✔	✔	✔	✔	✔	✔	✔
j (8 bits)	✔	✔	✔	✔	✔	✔	✔	✔
j2 (8 bits)	✗	✗	✗	✔	✔	✗	✗	✔
k_32 (8 bits)	✗	✗	✗	✗	✗	✗	✔	✗
k (8 bits)	✗	✗	✗	✗	✗	✗	✗	✔
a (8 bits)	✗	✗	✗	✗	✗	✗	✗	✔
w (8 bits)	✗	✗	✗	✗	✗	✗	✗	✔
Pipeline Registers: Name (width)								
IR (8 bits)	✔	✔	✔	✔	✔	✔	✔	✔
i (8 bits)	✔	✔	✔	✔	✔	✔	✔	✔
Sj (8 bits)	✔	✔	✔	✔	✔	✗	✗	✗
Si (8 bits)	✗	✗	✔	✔	✗	✔	✔	✔
Si_m(24 bits)	✗	✗	✗	✗	✗	✔	✔	✗
Memories: Name (depth)								
K (32 words)	✔	✔	✔	✔	✔	✔	✔	✔
IV (32 words)	✗	✔	✔	✗	✔	✗	✗	✗
S0 (256 words)	✔	✔	✔	✔	✔	✔	✔	✔
S1 (256 words)	✔	✔	✗	✔	✔	✔	✔	✔
S2 (256 words)	✗	✔	✗	✔	✗	✔	✔	✔
S3 (256 words)	✗	✔	✗	✔	✗	✔	✔	✔

Hence both *Si* and *Sj* are saved as pipeline registers to be used during swapping. For RC4A/RC4B *Si* is required in addition to *Sj* for ensuring two parallel RC4 execution. The two word variants of RC4, i.e., NGG and GGHN, a 32 bit PRGA word calculation requires a 24 bits *Si_m* pipeline register, other than reusing the 8 bit *Si* for the LSB of the 32 bit value.

- **Memory blocks reuse**: RC4 requires no IV memory. Hence for RC4-AccSuite when RC4$^+$ is accommodated in RC4-AccSuite, along with RC4, all the three memories i.e., K, S0 and S1 are reused along with the requirement of three new memories i.e., IV, S2 and S3. Similarly, Spritz has a 100% reuse of three memories used by RC4.

V. Implementation and Benchmarking

For experimentation and modeling of configurable RC4-AccSuite, an *incremental build* was followed to accommodate one additional RC4 variant at every step. For each design point, a pipelined processor design was optimized and developed, maximizing memory replication and resource reuse. The integrated configurable core of RC4-AccSuite executing RC4 and RC4$^+$ is termed as *RC4C-1*. Following similar nomenclature, the version after integration of VMPC was called *RC4C-2* and so on. Hence *RC4C-7* is the most flexible version of RC4-AccSuite, capable of being configured to execute any of the variants of RC4. Consequently a series of interesting design points were encountered, which were benchmarked for resource economization, power utilization and performance. Worth mentioning is that the user may choose *any number* or *any combination* of RC4 variants as per his requirements for RC4-AccSuite, *RC4C-1* till *RC4C-7* show only a trend of how area economizes and power budget rises.

All the designs were modeled using a high level processor description language LISA [35] (Synopsys PD version 2012.06-SP2). Synthesis was carried out with Synopsys design compiler, version 2009.06-SP4 using the Faraday standard cell libraries in *topographical mode*, technology node UMC SP/RVT Low-K process 65nm CMOS. For synthesizing the memory macros, Faraday memory compiler at 65nm technology node was used, the best case for memory modules with column multiplexer width 4 were recorded. We estimated the power consumption by Synopsys Power Compiler based on RTL switching activity annotation.

A. Throughput

The operating frequency of any version of RC4-AccSuite is determined by the time to access the largest memory in the memory bank, since the memory modules have a larger critical path than the core. For our design, the 256-word memories S0-S3 have access time of 0.7644 ns, indicating a maximum operating frequency of 1.3GHz. The throughput of individual cores of RC4 variants as well as throughput in any of the combination version of RC4-AccSuite is the same and is indicated by Table X.

The SI, KSA and IVSA phases are together named as keystream *Initialization*. Due to the use of dual ported SRAMs, 256 bytes/words initialization require no more than 128 cycles for SI. During KSA and IVSA, intermediate *nops* are put in the assembly instructions. For RC4, 2 *nops* are inserted between all consecutive 256 KSA instructions. RC4$^+$ has 3 layers of KSA, each having 256 instructions. For VMPC both KSA and an optional IVSA take 768 cycles with 7 *nops* in between consecutive instructions. RC4b has equal randomization sessions with KSA an IVSA, while GGHN requires 20 iterations of scrambling. For Spritz, a 32-byte key requires 64 *absorb_nibble* instructions (with one *nop* after every instruction). When shuffle is required, Whip and Crush of the state is called 3 and 2 times, respectively, requiring an additional 5.3μs as initialization time.

TABLE X
THROUGHPUT PERFORMANCE OF VARIOUS RC4 VARIANTS ON RC4-ACCSUITE (C/B IS CYCLES/BYTE)

Variant Core	SI (cycles)	KSA+IVSA (cycles)	Initialization (μs)	Throughput C/B	Throughput (Gbps)
RC4	128	256×2	0.39	2	5.20
RC4$^+$	128	256×3×2	1.18	2	5.20
VMPC	128	768×2×7	8.27	7	1.49
RC4A/B	128	256×2	0.39	1	10.40
RC4b	128	256×3×2	1.18	2	5.20
NGG	128	256×2	0.39	0.75	13.87
GGHN	128	256×2×20	7.88	0.75	13.87
Spritz	128	32×2×2 +(512×4×3) +(128×3×2)	0.098 +5.31	4	2.6

An interesting observation is that the throughput of RC4 and RC4$^+$ (after memory replication) is the same, in-spite of the added security margin for the later. The parallelization for RC4A and RC4B doubles the throughput compared to RC4. The word variants have the highest throughput performance since they generate a 32 bit word per PRGA instruction.

B. Area

The core area estimates of sequential as well as combinational logic, along with the memory area for different incremental versions of RC4-AccSuite are specified in Table XI in terms of equivalent NAND gates. The first half of the table specifies area of the individual pipelined cores of RC4-AccSuite while the second half refers to the configurable combination versions. Dual ported, byte-wide SRAMs were considered having 32 words for K and IV memories (4.609 KGE for each) and 256 words for S0-S3 (7.889 KGE for each). The total area is clearly dominated by the memory area.

The extent of resource sharing and a consequential core area economization can be visualized in Fig. 7. The area of a single algorithm core and a configurable RC4C-x core is compared against the sum of area of single algorithm cores that this version is able to support, i.e., the sum of RC4 and RC4$^+$ cores are added up and compared with the RC4C-1 core area and is found to be 12.3% less due to aggressive resource reuse. This *area economization margin* increases as more flexible versions of RC4C-x are analyzed, from left to right in Fig. 7. Hence for RC4C-7 this margin grows to reach 41.12%, justifying the need and rationale of developing configurable cores.

A noteworthy point is that the area economization calculation for RC4-AccSuite core excludes the memory bank contribution, considering that the area saving for RC4C-7

TABLE XI
AREA (KGE) FOR RC4-ACCSUITE VERSIONS

RC4-AccSuite Version	Core Area			Memory Area	Total Area
	Comb.	Seq.	Total		
RC4	0.43	0.48	0.914	20.387	21.301
RC4^{+}	1.158	0.468	1.627	40.774	42.401
VMPC	1.080	1.249	2.329	17.107	19.436
RC4A/B	0.996	0.534	1.530	36.165	37.695
RC4b	1.705	0.718	2.423	24.996	27.419
NGG	2.216	1.087	3.303	36.165	39.468
GGHN	2.164	1.418	3.582	36.165	39.747
Spritz	1.490	0.847	2.336	20.387	22.723
RC4C-1	1.534	0.694	2.228	40.774	43.002
RC4C-2	2.644	1.805	4.450	40.774	45.224
RC4C-3	2.967	2.181	5.149	40.774	45.923
RC4C-4	4.121	2.172	6.293	40.774	47.067
RC4C-5	4.392	2.907	7.299	40.774	48.073
RC4C-6	6.012	3.236	9.249	40.774	50.023
RC4C-7	8.654	2.904	11.558	40.774	52.332

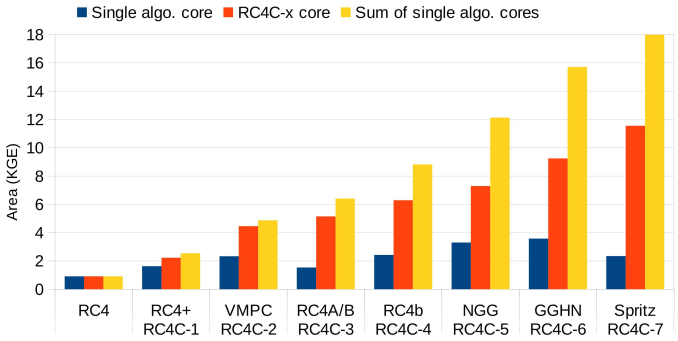

Fig. 7. RC4-AccSuite core Area economization

reaches up-to 79%. Since multiple arbiters time-share large memory banks that are extensively used in modern heterogeneous systems, for resource budgeting it is fair to consider only the core of a crypto-processor. Moreover, for most of the FPGA and CGRA like coarse grained hardware platforms, memories with desired size can be configured using block RAM modules that are available as macros.

C. Power

The power consumption of a design is a function of its complexity/flexibility and the clock frequency. Table XIII specifies the increasing core power consumption of the same algorithm, i.e., RC4 when run on various versions of RC4-AccSuite at 1.3 GHz. The user may keep the power budget minimal by choosing a version with flexibility no more than required. Faraday memory compiler reports the dynamic power of 32 and 256-byte memory to be 4.94 pJ/access and 5.53 pJ/access, respectively. Since these memory power budgets are not dependent on the design of the core, these values are not included in the total core power calculation in Table XIII.

A similar trend is seen from Fig. 8, showing the core dynamic power of RC4-AccSuite versions for different variants. All the power estimations include the initialization of stream ciphers and the generation of 1024 bits of keystream. The lower power budget utilized by VMPC is due to its least unshared resources, i.e., 7 stage datapath pipeline while none of the rest of the variants need more than 4.

TABLE XIII
THE POWER CONSUMPTION (DYNAMIC AND STATIC) BY RC4 ALGORITHM WHEN RUN ON VARIOUS RC4-ACCSUITE CORES

RC4-AccSuite Version	Static (μW)		Dynamic (mW)	Total (mW)
	Memory	Core	Core	Core
RC4	6.27	6.27	726.57	726.576
RC4C-1	12.53	12.49	1,107.90	1,107.91
RC4C-2	12.53	19.94	2,052.70	2,052.72
RC4C-3	12.53	21.09	2,358.70	2,358.72
RC4C-4	12.53	28.05	2,444.90	2,444.92
RC4C-5	12.53	34.26	3,274.40	3,274.43
RC4C-6	12.53	45.10	3,754.30	3,754.34
RC4C-7	12.53	65.48	6,571.00	6,571.06

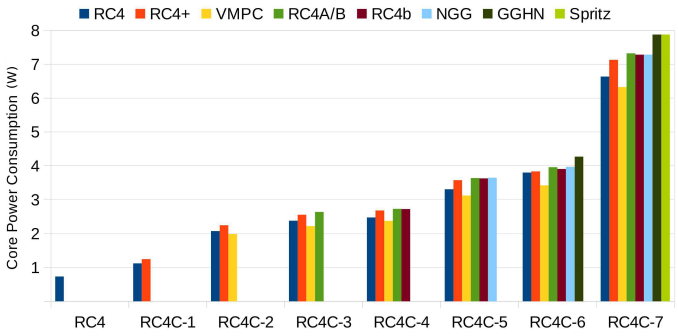

Fig. 8. Core dynamic power consumption in RC4-AccSuite versions

D. Comparison with Hardware Performance

We compare our implementations to the best known SRAM based hardware implementations of RC4 variants against area-efficiency (throughput per area, TPA), as reported in Table XII. Though FPGA and ASIC implementations cannot be fairly compared, the fastest FPGA implementations are mentioned in the table for reference. A configurable core supporting both RC4 and a less computationally intensive version of RC4^{+}, i.e., PRGA$^{\alpha}$ is reported in [1]. It is the fastest CMOS implementation of RC4 and also reports a 2 cycles/byte throughput. However, due to aggressive resource sharing and memory replication, RC4C-1 justifiably outperforms in area-efficiency. Their storage class memory (SCM) based RC4 implementation results in an encryption speed of 3.24 Gbps with 22.09 KGE of area resulting in a TPA of 0.30. The fastest CMOS based RC4 implementation on a comparable technology using SCMs for S-boxes reports 17.76 Gbps with an area of 50.58 KGE [32]. This design has a better TPA (0.35) than our implementation however does not set a good framework for flexibility extensions. Firstly because of its large area budget primarily due to numerous access ports per SCM, which may not be fully utilized of other RC4 variants and secondly because of the unlikely re-usability of the algorithm specific data coherency checks logic.

A CMOS implementation for no other RC4 variants is reported. For RC4A, an Altera FPGA implementation is reported with 0.18 Gbps of throughput performance [36], that is around 57× slower than our RC4A implementation. What remains incomparable is the extent of area economization due to resource re-usability due to absence of flexible, configurable RC4-AccSuite versions that have not been taken up for hardware implementation before.

TABLE XII
AREA COMPARISON OF RC4-ACCSUITE VERSIONS (C/B= CYCLES/BYTE, TPA = THROUGHPUT/ AREA)

Variant Name	Implementation Platform	Freq. MHz	Area KGE	KSA+IVSA cycles	KSA+IVSA μs	Throughput C/B	Throughput Gbps	TPA Mbps/GE
RC4 [31]	Xilinx XC4kE	160	-	642	0.79	2	0.64	-
RC4 [1]	65 nm CMOS	810	22.09	512	0.63	2	3.24	0.15
RC4 (this work)	65 nm CMOS	1300	21.30	512	.39	2	5.20	0.24
RC4, KSA+, PRGA$^{\alpha}$ [1]	65 nm CMOS	810	35.73	1536	1.90	2	3.24	0.09
RC4C-1 (this work)	65 nm CMOS	1300	45.22	1536	1.18	2	5.20	0.11
RC4A [36]	APEXTM 20K200E	33.33	-	2042	61.27	1.5	0.18	-
RC4A (this work)	65 nm CMOS	1300	37.70	512	.39	1	10.40	0.28

TABLE XIV
RC4-ACCSUITE VERSIONS SOFTWARE PERFORMANCE

RC4 Variant	Platform	KSA+IVSA cycles	KSA+IVSA μs	PRGA cycles	PRGA Gbps
RC4 [7]	Pentium 4 ,	16945	6.05	14	1.56
RC4^{+} [7]	2.8 GHz, 512 MB DDR	49824	17.79	25	0.91
VMPC [11]	Pentium 4, 2.66 GHz	8580	3.2	13	1.68
NGG [15]	32-bit PC	-	-	-	4.83
GGHN [16]		-	-	-	4.98
Spritz [8]	Macbook Air	-	-	24	0.09

E. Comparison with Software Performance

Software performance on general purpose computers for various RC4 variants is tabulated in Table XIV. RC4-AccSuite renders initialization for RC4 and RC4^{+} that is about $15\times$ faster than the one reported in [7]. Similarly, RC4-AccSuite performance for NGG and GGHN is more than $4.3\times$ faster than their respective references [15], [16]. Their initialization time is not specified for comparison. Spritz on RC4-AccSuite has PRGA performance that is $27\times$ faster than the one reported on a Macbook Air (1.8GHz Core i5) [8]. For VMPC on RC4-AccSuite, the KSA and PRGA phases are comparable to the reported software performance [11] due to the high dependence of sequential memory accesses in VMPC function, rendering slow performance due to *nops* between two PRGA instructions.

VI. CONCLUSION AND OUTLOOK

In the context of flexible yet efficient cryptographic accelerators for stream ciphers, this work studies the design of RC4-AccSuite, a configurable co-processor for the family of RC4-like ciphers. Its flexibility stands out due to its ability to switch to another algorithm on-the-fly as per the user requirements of throughput, power or security changes while its rich instruction set can be used to map newer RC4 variants. RC4-AccSuite significantly stands out in its area-efficiency against SRAM based dedicated hardware accelerators for stream ciphers. The detailed physical design of the processor is part of our future road-map. The idea of resource reuse for common kernel cryptographic algorithms will be further probed for other classes of similar block ciphers.

REFERENCES

[1] A. Chattopadhyay and G. Paul, "Exploring security-performance trade-offs during hardware accelerator design of stream cipher RC4," in *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, pp. 251–254, IEEE, 2012.

[2] P. Sepehrdad, P. Susil, S. Vaudenay, and M. Vuagnoux, "Tornado attack on RC4 with applications to WEP & WPA," *IACR Cryptology ePrint Archive*, vol. 2015, p. 254, 2015.

[3] M. Vanhoef and F. Piessens, "All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. (J. Jung and T. Holz, eds.), pp. 97–112, USENIX Association, 2015.

[4] R. Basu, S. Maitra, G. Paul, and T. Talukdar, "On some sequences of the secret pseudo-random index j in rc4 key scheduling," in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pp. 137–148, Springer, 2009.

[5] A. Maximov and D. Khovratovich, "New state recovery attack on rc4," in *Advances in Cryptology–CRYPTO 2008*, pp. 297–316, Springer, 2008.

[6] A. Popov, "Prohibiting RC4 Cipher Suites." RFC 7465, Oct. 2015.

[7] S. Maitra and G. Paul, "Analysis of RC4 and proposal of additional layers for better security margin," in *Progress in Cryptology-INDOCRYPT 2008*, pp. 27–39, Springer, 2008.

[8] R. L. Rivest and J. C. Schuldt, "Spritz–A spongy RC4-like stream cipher and hash function." CRYPTO 2014 Rump Session, 2014.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak sponge function family main document," *Submission to NIST (Round 2)*, vol. 3, p. 30, 2009.

[10] B. Schneier, *Applied Cryptography*, ch. 17, pp. 397–398. John Wiley and Sons, 1996.

[11] B. Zoltak, "VMPC one-way function and stream cipher," in *Fast Software Encryption*, pp. 210–225, Springer, 2004.

[12] S. Paul and B. Preneel, "A new weakness in the rc4 keystream generator and an approach to improve the security of the cipher," in *Fast Software Encryption*, pp. 245–259, Springer, 2004.

[13] J. Lv, B. Zhang, and D. Lin, "Distinguishing Attacks on RC4 and A New Improvement of the Cipher." Cryptology ePrint Archive, Report 2013/176, 2013. http://eprint.iacr.org/2013/176.

[14] M. McKague, "Design and Analysis of RC4-like Stream Ciphers." Master Thesis, 2005.

[15] Y. Nawaz, K. C. Gupta, and G. Gong, "A 32-bit RC4-like Keystream Generator.," *IACR Cryptology ePrint Archive*, vol. 2005, p. 175, 2005.

[16] G. Gong, K. C. Gupta, M. Hell, and Y. Nawaz, "Towards a general RC4-like keystream generator," in *Information Security and Cryptology*, pp. 162–174, Springer, 2005.

[17] C. Boura, M. Naya-Plasencia, and V. Suder, "Scrutinizing and improving impossible differential attacks: Applications to clefia, camellia, lblock and simon," in *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I* (P. Sarkar and T. Iwata, eds.), vol. 8873 of *Lecture Notes in Computer Science*, pp. 179–199, Springer, 2014. Full version avialble at https://eprint.iacr.org/2014/699.

[18] C.-P. Su, C.-L. Horng, C.-T. Huang, and C.-W. Wu, "A configurable AES processor for enhanced security," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pp. 361–366, ACM, 2005.

[19] A. Satoh and S. Morioka, "Unified hardware architecture for 128-bit block ciphers AES and Camellia," in *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 304–318, Springer, 2003.

[20] S. Sen Gupta, A. Chattopadhyay, and A. Khalid, "HiPAcc-LTE: an integrated high performance accelerator for 3GPP LTE stream ciphers," in *Progress in Cryptology–INDOCRYPT 2011*, pp. 196–215, Springer, 2011.

[21] S. S. Gupta, A. Chattopadhyay, and A. Khalid, "Designing integrated accelerator for stream ciphers with structural similarities," *Cryptography and Communications*, vol. 5, no. 1, pp. 19–47, 2013.

[22] M. Rogawski, K. Gaj, and E. Homsirikamol, "A high-speed unified hardware architecture for 128 and 256-bit security levels of AES and the SHA-3 candidate grøstl," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 37, no. 6-7, pp. 572–582, 2013.
[23] K. Järvinen, "Sharing resources between AES and the SHA-3 second round candidates Fugue and Grøstl," in *The Second SHA-3 Candidate Conference*, p. 2, 2010.
[24] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: a fast flexible architecture for secure communication," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 110–119, IEEE, 2001.
[25] R. Buchty, N. Heintze, and D. Oliva, *Cryptonite–A programmable crypto processor architecture for high-bandwidth applications*. Springer Berlin Heidelberg, 2004.
[26] D. Theodoropoulos, I. Papaefstathiou, and D. Pnevmatikatos, "Cproc: An efficient Cryptographic Coprocessor," in *16th IFIP/IEEE International Conference on Very Large Scale Integration*, pp. 1–4, IEEE, 2008.
[27] K. Shahzad, A. Khalid, Z. E. Rákossy, G. Paul, and A. Chattopadhyay, "CoARX: a coprocessor for ARX-based cryptographic algorithms," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–10, IEEE, ACM, 2013.
[28] P. Yalla, E. Homsirikamol, and J.-P. Kaps, "Comparison of multi-purpose cores of keccak and aes," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 585–588, EDA Consortium, 2015.
[29] P. Kitsos, G. Kostopoulos, N. Sklavos, and O. Koufopavlou, "Hardware implementation of the RC4 stream cipher," in *Circuits and Systems, 2003 IEEE 46th Midwest Symposium on*, vol. 3, pp. 1363–1366, IEEE, 2003.
[30] D. Matthews, "Methods and apparatus for accelerating arc4 processing." US Patent Number 7403615, Morgan Hill, CA, July, 2008, July 2008.
[31] T. H. Tran, L. Lanante, Y. Nagao, M. Kurosaki, and H. Ochi, "Hardware implementation of high throughput RC4 algorithm," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pp. 77–80, IEEE, 2012.
[32] S. S. Gupta, A. Chattopadhyay, K. Sinha, S. Maitra, and B. P. Sinha, "High-performance hardware implementation for RC4 stream cipher," *Computers, IEEE Transactions on*, vol. 62, no. 4, pp. 730–743, 2013.
[33] A. Chattopadhyay, A. Khalid, S. Maitra, and S. Raizada, "Designing high-throughput hardware accelerator for stream cipher HC-128," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pp. 1448–1451, IEEE, 2012.
[34] A. Khalid, P. Ravi, A. Chattopadhyay, and G. Paul, "One Word/Cycle HC-128 Accelerator via State-Splitting Optimization," in *Progress in Cryptology–INDOCRYPT 2014*, pp. 283–303, Springer, 2014.
[35] A. Chattopadhyay, H. Meyr, and R. Leupers, "LISA: a uniform ADL for embedded processor modelling, implementation and software toolsuite generation," *Processor Description Languages*, vol. 1, pp. 95–130, 2008.
[36] A. Al Noman, R. Sidek, L. Ali, *et al.*, "RC4A stream cipher for WLAN security: A hardware approach," in *Electrical and Computer Engineering, 2008. ICECE 2008. International Conference on*, pp. 624–627, IEEE, 2008.

AUTHOR BIOGRAPHIES

Ayesha Khalid completed her B.E. in Computer Systems Engineering from National University of Sciences and Technology (NUST), Pakistan. She did her M.S. in Electrical Engineering from Center for Advanced Studies in Engineering (CASE), affiliated with University of Engineering and Technology, UET-Taxila, Pakistan. From 2008 to 2010, she served as a Lecturer in the Department of Electrical Engineering at Muhammad Ali Jinnah University, Islamabad and later joined RWTH Aachen, Germany as a Research Associate for her doctoral studies. She is the recipient of DAAD scholarship award for Ph.D. Her dissertation focuses on the identification, characterization and exploitation of representative cryptographic operations/ structures for a whole class of cryptography, enabling high-level synthesis of cryptographic proposals. Currently, she is working as a Research Fellow at Queens University Belfast (QUB) in the SAFEcrypto project.

Goutam Paul completed his undergraduate in Computer Engineering in 2001 from Bengal Engineering College (Deemed University), now Indian Institute of Engineering Science and Technology (IIEST), Shibpur, Howrah, India; Master degree in Computer Science in 2003 from State University of New York (SUNY) at Albany, U.S.A.; and Ph.D. in Cryptology in 2009 from Indian Statistical Institute, Kolkata (degree awarded from Jadavpur University, Kolkata, India). From 2006 to 2013, he was an Assistant Professor in the Department of Computer Science and Engineering of Jadavpur University and during 2012-2013, he visited RWTH Aachen, Germany as a Humboldt Fellow. From August 2013, Goutam Paul has been serving Indian Statistical Institute, Kolkata, as an Assistant Professor. He also received the Young Scientist Award from the National Academy of Sciences, India (NASI) in 2013. His doctoral research focussed on the analysis of RC4, the then most popular and most widely commercially deployed software stream cipher and also the then standard encryption tool for IEEE WiFi protocol; and his work revealed many critical weaknesses of the cipher and initiated a chain of subsequent research by others in this area. Later he also worked on the analysis of other stream ciphers like HC-128, Grain-v1, Salsa20. Recently, he has taken up keen interest in efficient hardware design of cryptographic primitives and in the analysis of BB84-like quantum key distribution protocols. Goutam Paul is the author of one book and more than 60 papers in peer-reviewed international journals and conferences. He regularly serves as the TPC member of many top conferences, reviewer of many premier journals and presents invited seminars in internationally acclaimed venues. He is a member of ACM and a senior member of IEEE.

Anupam Chattopadhyay received his B.E. degree from Jadavpur University, India in 2000. He received his MSc. from ALaRI, Switzerland and PhD from RWTH Aachen in 2002 and 2008 respectively. From 2008 to 2009, he worked as a Member of Consulting Staff in CoWare R&D, Noida, India. From 2010 to 2014, he led the MPSoC Architectures Research Group in RWTH Aachen, Germany as a Junior Professor. Since September, 2014, he is appointed as an assistant Professor in SCE, NTU, Singapore. During his PhD, he worked on automatic RTL generation from the architecture description language LISA, which was commercialized later by a leading EDA vendor. In his academic stint, he is active in domain-specific high-level synthesis for cryptography, high-level reliability estimation flows, generalization of classic linear algebra kernels and coarse-grained reconfigurable architectures. In these areas, he published as a (co)-author over 80 conference and journal papers, several book-chapters and a book. Anupam served in several TPCs of top conferences, regularly reviews journal articles and presented multiple invited seminars in prestigious venues. He is a member of ACM and a senior member of IEEE.

APPENDIX

This is a substantially revised and extended version of the conference paper [1], authored by Anupam Chattopadhyay (third author of the current draft) and Goutam Paul (second

author of the current draft), that was accepted in VLSI-SoC 2012. Below we summarize the important differences between the current draft and the earlier published draft.

1) The conference paper talks about a reconfigurable processor capable of stream cipher encryption by RC4 and its one variant RC4+. This journal version extends the idea for a number of RC4 variants, i.e., VMPC, RC4A, RC4B, RC4b, NGG, GGHN and Spritz.
2) The conference paper lacked a conscious effort/ discussion of resource reuse for similar cores, except for where an entire instruction/ memory could be reused. The current version takes on an extensive reuse economization of combinational recourses (entire or partial instruction datapath reuse as discussed in Section 4.3.1 and 4.3.2) and sequential recourses (registers/ memories reuse). Consequently, RC4-AccSuite extensively reuses resources saving up to 41% in terms of area, compared to individual cores, with power budget dictated primarily by the variant used (discussed in Section 5.2).
3) For the memory replication undertaken in the conference paper for RC4+, a replication factor of 2 is considered for RC4-$^{\alpha}$ (a lighter version of RC4+). The current version takes up the replication of memories for RC4+ instead up to a factor of 3. Consequently, the throughput improves too.
4) The current work did not start from the core presented in the conference version, all the work here has been taken up from scratch. Consequently, the results stand out for RC4 and RC4+, and for other variants are entirely new.